Practice Session 08 : CI/CD & Continuous Testing

Make sure that you have already gone through Lab-07.

In this session, you will focus on the continuous testing in GitLab.
Continuous Testing: Continuous Testing is the process of executing automated tests as part of the software delivery pipeline in support of better speed and efficiency when managing deployments.
Continuous Testing in GitLab: You can configure your gitlab ci pipeline with the testing jobs using different tools provided by GitLab

In this practice session, you will write functions in python that will do several operations related to the csv file. Recall the practice session, where you have prepared the python code that reads a csv file and writes the output to a webpage.

In addition to above, you will also perform a code quality test. The code quality report will indicate the places in the python files, where your code can further be improved.

This practice session has two parts:

1. Testing your code using python testing frameworks:
    a. Here you will first write several very simple python functions
    b. You will test the functions using python's `unittest` framework.
    c. You will also test the function using `pyTest` framework and `assert` keyword
    d. Here you will also see the code coverage using `coverage` tool
    e. In this part you will only use the python environment in a newly created VM.
        i. You may use your own laptop instead of a VM in ETAIS
2. Continuous testing using GitLab
    a. Here you will first upload the above code including the testing codes to your gitlab repo.
    b. Create and checkout a new branch
    c. Configure the GitLab CI configuration file in the newly created branch
        i. To see the code quality report using codeclimate tool
        ii. To see the test report
        iii. To see the test coverage

Prerequisite

- Basic knowledge on Python, pandas library, csv file format
- Basic knowledge on YAML
- Familiar with Dockerfile
- Familiar with Git

0. Preparing the Environment:

0.1. Install necessary Software

Please login to the `k8s-controller` VM. Make sure that the following software packages are installed. If you have finished Lab-07, it is expected the below software packages are installed.

- Python3.8 or later :
- Pip : version 22.3
    - `pip --version` to verify the installation and version.
- `pandas` module: `pip show pandas` to verify the installation and version
- Docker
- gitlab-runner
    - Don't forget to add `gitlab-runner` user to the `docker` group using `usermod -aG docker gitlab-runner` in your VM.
    - `gitlab-runner -v` to verify the installation and version.

0.2. Gitlab Project

- Create a blank project in the gitlab with name `lab08-cicd-testing` under your group `Devops2022fall/students/devops2022Fall-<lastname>-<studyCode>`

0.3. Notes

Below some of the instructions are descriptive. That means you may get errors while executing the given minimal version of the commands. You need to investigate and fix those errors. For this, you may need to google and debug the error by yourself.

PART-1 : Testing your code using python testing frameworks

In this part, we assume that you are in your `k8s-controller` VM. However, you may perform this part (PART-1) on only your laptop/PC.

STEP-1.1:

1. Clone the newly created project to the home directory.
2. Change directory to `$HOME/lab08-cicd-testing` folder
3.
4. Create following directories: `mkdir csv_data src unit-test`

Directory Description:

- `csv_data` : This directory is for the csv data. Recall the csv file that you have used in the previous practice session. You can copy that csv file to `csv_data` directory OR you can simply download and reuse the files available **https://gitlab.cs.ut.ee/devops22fallpub/lab05-shared-microservice/-/raw/ff0e3c87f83cf37e60f7727c57ea74928bee83ea/CO2.csv** and no need to use any other csv files, even in later steps.
- `src` : All the python code will be kept here.
- `unit-test` : The python code for testing the source files will be kept here.

Commit with the message “STEP-1.1: created necessary folders and co2.csv file” and push the above changes.

STEP-1.2:

Lets create a python file with the required functions to read the csv and perform some time conversion tasks.

1. Create `analyze_csv.py` python file inside `src` folder with following functions:

Filename: `analyze_csv.py`

```
import pandas as pd
import os
import datetime

# tested on 13 digit input
def convert_epoch_to_human_readable(epochTime):
    hr_date =
datetime.datetime.fromtimestamp((epochTime/1000.0)).strftime('%Y-%m-%d
%H:%M:%S.%f')
    return hr_date
def get_year(epochTime):
    return datetime.datetime.fromtimestamp((epochTime/1000.0)).strftime('%Y')
def get_month(epochTime):
    return datetime.datetime.fromtimestamp((epochTime/1000.0)).strftime('%m')
def get_date(epochTime):
    return datetime.datetime.fromtimestamp((epochTime/1000.0)).strftime('%d')
def get_hour(epochTime):
    return datetime.datetime.fromtimestamp((epochTime/1000.0)).strftime('%H')
def get_minutes(epochTime):
    return datetime.datetime.fromtimestamp((epochTime/1000.0)).strftime('%M')
def get_seconds(epochTime):
    return datetime.datetime.fromtimestamp((epochTime/1000.0)).strftime('%S')
def get_miliseconds(epochTime):
    return datetime.datetime.fromtimestamp((epochTime/1000.0)).strftime('%f')

# argument: @filenmae: path to the file in string format, e.g.
"./dehury/csv_data/co2.csv"
def read_csv(filename):
  df = pd.read_csv(filename)
  return df
```

```
# statement to read the csv file present in csv_data directory using read_csv
function.



# Invoke the above convert_epoch_to_human_readable() function with a random
epochTime, e.g. 1600763816518


# Invoke above get_* functions with the epochTime 1600763816518.
```

2. Now you need to update the `analyze_csv.py` file that do the followings:
    a. Invoke the above `read_csv()` function and print the returned dataframe. Pass the path to the csv file present in your `csv_data` directory.
    b. Invoke the above `convert_epoch_to_human_readable()` function with a random epochTime, e.g. 1600763816518 and print the returned value.
    c. Invoke above `get_*` functions with the `epochTime` 1600763816518 and print the returned values.
3. Remember, this updated `analyze_csv.py` will be used in later steps.

Commit with the message “STEP-1.2: created and updated analyze_csv.py file ” and push the above changes.

STEP-1.3:

Now it is time to test the above functions present in `analyze_csv.py` python file with some test cases.

1. Before proceeding further, create a csv file with empty data (only with the header) inside `csv_data` directory.

Filename and path: `csv_data/file_with_no_data.csv`

```
name,tags,time,host,unit,value
```

2. Create the following file inside the `unit-test` directory.

Filename: `test_analyze_csv.py`

```
import unittest
import sys
sys.path.append('./')
from src.analyze_csv import *

class test_analyze_csv(unittest.TestCase):

    def setUp(self):
        self.epochTime = 1600763816518
        self.file_with_data = "<path_to_co2_file>"
        self.file_no_data = "<path_to_the_file_with_no_data>"
```

```
    def test_read_csv_with_data(self):
        self.assertFalse(read_csv(self.file_with_data).empty)

    def test_read_csv_with_no_data(self):
        self.assertTrue(read_csv(self.file_no_data).empty)

    def test_convert_epoch_to_human_readable(self):
        self.assertEqual( convert_epoch_to_human_readable( self.epochTime ),
"2020-09-22 08:36:56.518000" )
    # def test_get_year(self):
        # add your test statement here
    # def test_get_month(self):
        # add your test statement here
    # def test_get_date(self):
        # add your test statement here
    # def test_get_hour(self):
        # add your test statement here
    # def test_get_minutes(self):
        # add your test statement here
    def test_get_seconds(self):
        self.assertEqual(get_seconds(self.epochTime),"56")
    def test_get_miliseconds(self):
        self.assertEqual(get_miliseconds(self.epochTime),"1800")


if __name__ == '__main__':
   unittest.main()
```

3. Update the path to the csv files in `setUp()` function.
4. In the `test_analyze_csv.py` file, we are first creating a subclass of `unittest.TestCase`. The function `test_read_csv_with_data()` checks if the `read_csv()` function in `analyze_csv.py` python file is returning a dataframe with some data. Similarly, the test function `test_read_csv_with_no_data()` checks if `read_csv()` in `analyze_csv.py` python file is returning an empty dataframe.

5. In the `setUp()` function, we have set the value of `epochTime` to `1600763816518`. The test `test_convert_epoch_to_human_readable()` function checks if the `convert_epoch_to_human_readable()` function in `analyze_csv.py` python file is returning the desired output.

6. Similarly, `test_get_seconds()` and `test_get_miliseconds()` function test the returned value of `get_seconds()` and `get_miliseconds()` functions.
7. Execute the `test_analyze_csv.py` file.
8. At this point, you should see that a test function is failing. **Investigate** by yourself and fix the error that may exist in `test_analyze_csv.py` or `analyze_csv.py` python file.
9. Now further update the `test_analyze_csv.py` file by adding the test statement in `def test_get_year()`, `def test_get_month()`, `def test_get_date()`, `def test_get_hour()`, and `def test_get_minutes()` functions.

Commit with the message "*STEP-1.3: created and updated file_with_no_data.csv, test_analyze_csv.py files.*" and push the above changes.

STEP-1.4:

Instead of using the `unittest` tool as in the above step, we can use the `pytest` tool. For this, it is not necessary to create a separate subclass. Rather we will use the available `assert` keyword.

1. Install `pytest` module using `pip` command.
2. Inside the `src` directory create `analyze_csv_with_test.py` file with the content of `analyze_csv.py` python file:
   ```
   cp analyze_csv.py analyze_csv_with_test.py
   ```
3. Now inside the `src` directory, you should have two files: `analyze_csv.py` and `analyze_csv_with_test.py` with the same content.
4. Let's modify the content of `analyze_csv_with_test.py` as given below:

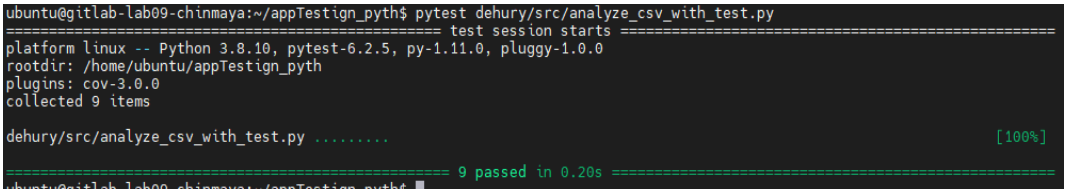
Filename and path: `src/analyze_csv_with_test.py` (below code is for your reference)

```
import pandas as pd
import os
import datetime

# tested on 13 digit input
def convert_epoch_to_human_readable(epochTime):
       hr_date =
datetime.datetime.fromtimestamp((epochTime/1000.0)).strftime('%Y-%m-%d
%H:%M:%S.%f')
       return hr_date

def get_year(epochTime):
       return
datetime.datetime.fromtimestamp((epochTime/1000.0)).strftime('%Y')
def test_get_year():
       assert get_year(testEpochTime) == "2020"

def get_month(epochTime):
       return
datetime.datetime.fromtimestamp((epochTime/1000.0)).strftime('%m')
def test_get_month():
  assert get_month(testEpochTime) == "09"

def get_date(epochTime):
       return
datetime.datetime.fromtimestamp((epochTime/1000.0)).strftime('%d')
def test_get_date():
  assert get_date(testEpochTime) == "22"

def get_hour(epochTime):
       return
datetime.datetime.fromtimestamp((epochTime/1000.0)).strftime('%H')
def test_get_hour():
  assert get_hour(testEpochTime) == "08"


def get_minutes(epochTime):
       return
datetime.datetime.fromtimestamp((epochTime/1000.0)).strftime('%M')
def test_get_minutes():
  assert get_minutes(testEpochTime) == "36"
```

```
def get_seconds(epochTime):
       return
datetime.datetime.fromtimestamp((epochTime/1000.0)).strftime('%S')
def test_get_seconds():
  assert get_seconds(testEpochTime) == "56"


def get_miliseconds(epochTime):
       return
datetime.datetime.fromtimestamp((epochTime/100.0)).strftime('%f')
def test_get_miliseconds():
  assert get_miliseconds(testEpochTime) == "1800"

# argument: @filenmae: path to the file in string format, e.g.
"./dehury/csv_data/co2.csv"
def read_csv(filename):
  df = pd.read_csv(filename)
  return df


# statement to read the csv file present in csv_data directory using read_csv
function.



# Invoke the above convert_epoch_to_human_readable() function with a random
epochTime, e.g. 1600763816518


# Invoke above get_* functions with the epochTime 1600763816518.
```

5. In the above, the yellow highlighted codes are meant for testing the corresponding functions. Notice that, we are not using any `unittest` tool to define the test functions, rather we are using the `assert` keyword.

6. Execute the above `analyze_csv_with_test.py` file using `pytest` command:
   `pytest <path_to_src_folder>` or
   `pytest <path_to_analyze_csv_with_test.py_file>`
   Here is the sample output:



7. All tests should pass. Fix the errors by yourself, if required.
8. Similar to the given test functions, add a test function for `read_csv()`. This should be similar to the test functions given in `test_analyze_csv.py` python file.

Note:

- Remember that `pytest` will search the csv file from your current location.
- You may also get errors in some other test functions. In that case the last few lines of the above `pytest` command may look like below:

```
                )
E               FileNotFoundError: [Errno 2] No such file or directory: './csv_data/CO2.csv'

../.local/lib/python3.8/site-packages/pandas/io/common.py:702: FileNotFoundError
=========================================== short test summary info ============================================
FAILED dehury/src/analyze_csv_with_test.py::test_get_miliseconds - AssertionError: assert '180000' == '18000'
FAILED dehury/src/analyze_csv_with_test.py::test_read_csv_with_data - FileNotFoundError: [Errno 2] No such file or direc...
========================================== 2 failed, 7 passed in 0.30s =========================================
```

SCREENSHOT:

Take the screenshot of the pytest output, similar to below:

```
[centos@k8s-controller-chinmaya lab08-cicd-testing]$ pytest ./src/analyze_csv_with_test.py
========================================== test session starts ==========================================
platform linux -- Python 3.9.9, pytest-7.1.3, pluggy-1.0.0
rootdir: /home/centos/lab08-cicd-testing
collected 9 items

src/analyze_csv_with_test.py .........                                                          [100%]

========================================== 9 passed in 0.27s ===========================================
[centos@k8s-controller-chinmaya lab08-cicd-testing]$
```

Keep the screenshot inside project's root directory with the name screenshot-step-1.4

Commit with the message "STEP-1.4: tested the code with unitest and pytest modules and uploaded the screenshot." and push the above changes.

Note:

- Remember that `pytest` will search the csv file from your current location.
- You may also get errors in some other test functions. In that case the last few lines of the above `pytest` command may look like below:

```
                )
E               FileNotFoundError: [Errno 2] No such file or directory: './csv_data/CO2.csv'

../.local/lib/python3.8/site-packages/pandas/io/common.py:702: FileNotFoundError
=========================================== short test summary info ============================================
FAILED dehury/src/analyze_csv_with_test.py::test_get_miliseconds - AssertionError: assert '180000' == '18000'
FAILED dehury/src/analyze_csv_with_test.py::test_read_csv_with_data - FileNotFoundError: [Errno 2] No such file or direc...
========================================== 2 failed, 7 passed in 0.30s =========================================
```

PART-2 : Continuous testing using GitLab

In PART-1, you tested the python function using `unittest` and `pytest` tools. However, in real world applications, you always host the code in private or public VCS platforms, such as GitHub or GitLab. You incorporate the test functions in VCS in a way that the tests are automatically performed when there is a commit or merge/pull request. In this part, we will see how the tests that you performed in PART-1 can be automatically invoked when there is a commit or merge request.

Make sure that you have finished PART-1 and your project `lab08-cicd-testing` is up-to-date.

STEP-2.0:

- Now you need to create and register the required gitlab-runners with following configurations. Please see the command below for the second gitlab-runner.
    - First Runner (should be in `k8s-controller` VM):

        - Tag: `pytest`
        - Runner: `shell`
    - [Optional] Second runner (should be created in your personal laptop where the at least 11GB of storage available):

```
sudo gitlab-runner register --executor "docker" \
    --docker-image="docker:stable" \
    --url "https://gitlab.cs.ut.ee/" \
    --description "codeclimate" \
    --tag-list "codeclimate" \
    --locked="false" \
    --access-level="not_protected" \
    --docker-volumes "/cache"\
    --docker-volumes "/var/run/docker.sock:/var/run/docker.sock" \
    --registration-token="<your registration token>" \
    --builds-dir /tmp/builds \
    --docker-volumes /tmp/builds:/tmp/builds \
    --non-interactive
```

- You may delete the old gitlab-runners (from `k8s-controller` VM) and create the above or you may reuse the old gitlab-runners. At this point, you should have at least one runner in the `k8s-controller` VM, with tag `pytest` and another optional one with tag `codeclimate` (if you have created this).
- `codeclimate` runner will be mainly used for checking the code quality and `pytest` will be used for other purposes.

Note: If you have skipped creating the second runner, don't worry. We have created one shared runner for you (with the tag `codeclimate-shared-devops`).

STEP-2.1:

In the previous step, we were invoking the `unitest` and `pytest` command from the CLI to test the python functions. In this step, we will perform the same testing operations inside a docker container.

- First step is to create a `docker` folder inside the project's root directory.
- Create a `DockerfileUnitTest` file inside `docker` directory with following content

Filename: `docker/DockerfileUnitTest` (sample code)

```
FROM python:3

WORKDIR /usr/src/app

RUN pip install --no-cache-dir pytest pandas


COPY ./ .

RUN mkdir report

CMD ["pytest", "--junitxml=./report/test_report.xml", "./src", "./unit-test"]
```

- The above code may need modification and may not work if you directly use this. Go through each line and try to update wherever it is required.

- Make sure that you have updated your `DockerfileUnitTest` file according to your directory structure.
- The last `CMD` line will export the test report `test_report.xml` in a XML format. This XML format can be easily understandable by the gitlab ci server.

Commit with the message “STEP-2.1: created and updated docker folder and the DockerfileUnitTest.” and push the above changes.

STEP-2.2:

- Now it is the time to create your first GitLab CI configuration file `.gitlab-ci.yml` inside the project’s root directory. Add the following content to the `.gitlab-ci.yml` configuration file.

Filename: `.gitlab-ci.yml`

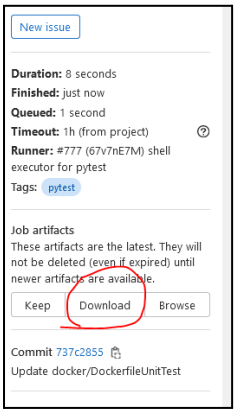
```
variables:
       PYTEST_IMAGE_NAME :
gitlab.cs.ut.ee:5050/dehury/gitlab-cont-testing-project/lab09pytest:latest

stages:
       - test

test_report_using_pytest:
  stage: test
  tags:
       - pytest
  script:
       - docker login -u dehury -p $gitlabpassword gitlab.cs.ut.ee:5050
       - docker build -t $PYTEST_IMAGE_NAME -f ./docker/DockerfileUnitTest
.
       - docker run -v $PWD:/usr/src/app/report/ $PYTEST_IMAGE_NAME

  artifacts:
       when: always
       expose_as: 'Test Report'
       paths: [test_report.xml]
       reports:
       junit: test_report.xml
```

- `$gitlabpassword` should be created in Gitlab `Settings` --> `CICD` --> Expand `Variables` with the deploy token as value (similar to the steps you followed in Lab-07)
- *IMP:* You may need to update the Yellow highlighted lines.

Commit with the message “STEP-2.2: created and updated gitlab ci file.” and push the above changes.

- Once committed and pushed to the `main` branch of your gitlab project, Go to `CI/CD` --> `Jobs` to see the list of jobs that are recently created by the gitlab ci pipeline.
    - Click on the name of the recent job
    - In the job log, you will see the detailed and short test summary info.
    - The test report is uploaded to the gitlab CI server as an artifact with the file name `test_report.xml` that can be downloaded once the job is finished.

- In the right hand side, you will see the options to download and browse the `test_report.xml` artifact, as shown in the figure below.



STEP-2.3: Find the test coverage

Test coverage tells the percentage of your codes that are covered by the test functions.

- Create a new branch `cicd_test_coverage` from the main branch. You can do so in GitLab web UI `Repository` --> `Branches` option.
- In `cicd_test_coverage` branch, create a new `DockerfileTestCoverage` with the following content. You may use the `Web IDE` interface to create and edit the files.

Filename: `docker/DockerfileTestCoverage` (sample code)

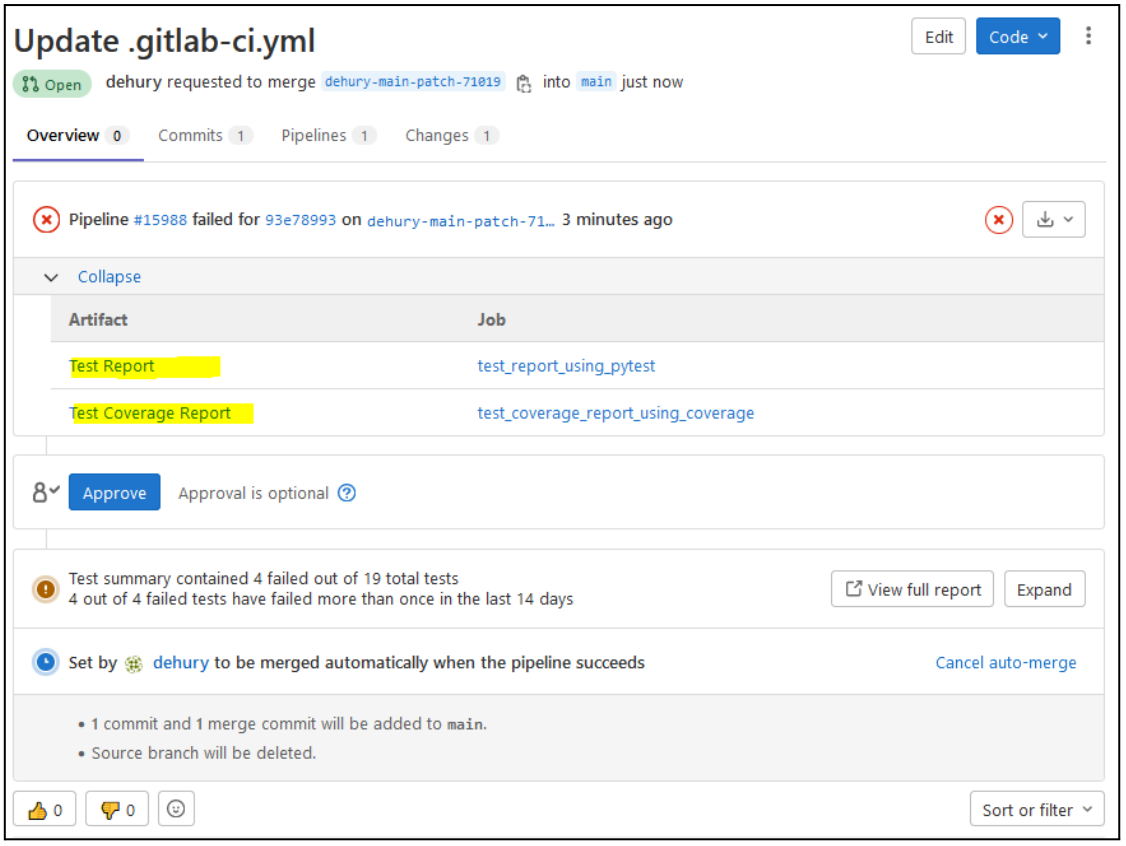
```
FROM python:3

WORKDIR /usr/src/app


RUN pip install --no-cache-dir pytest pandas pytest-cov

COPY ./ .

RUN mkdir report

RUN coverage run -m pytest || true

RUN coverage report

CMD ["coverage", "xml", "-o", "./report/coverage_report.xml",
"--omit=/usr/lib/*"]
```

- Similar to `DockerfileUnitTest` file, this docker file also uses `python:3` as the base image.
- The command that will be used to find the test coverage is `coverage`. For this notice in the third line, we are installing the `pytest-cov` tool using `pip` package manager.
- `coverage` module uses `pytest` module (as you have used in PART-1) to find the test coverage.
- `coverage report` command would allow you to see the coverage report in the GitLab job logs.
- The same coverage report will also be exported in xml format that is understandable by the GitLab CI server, as in the last line.

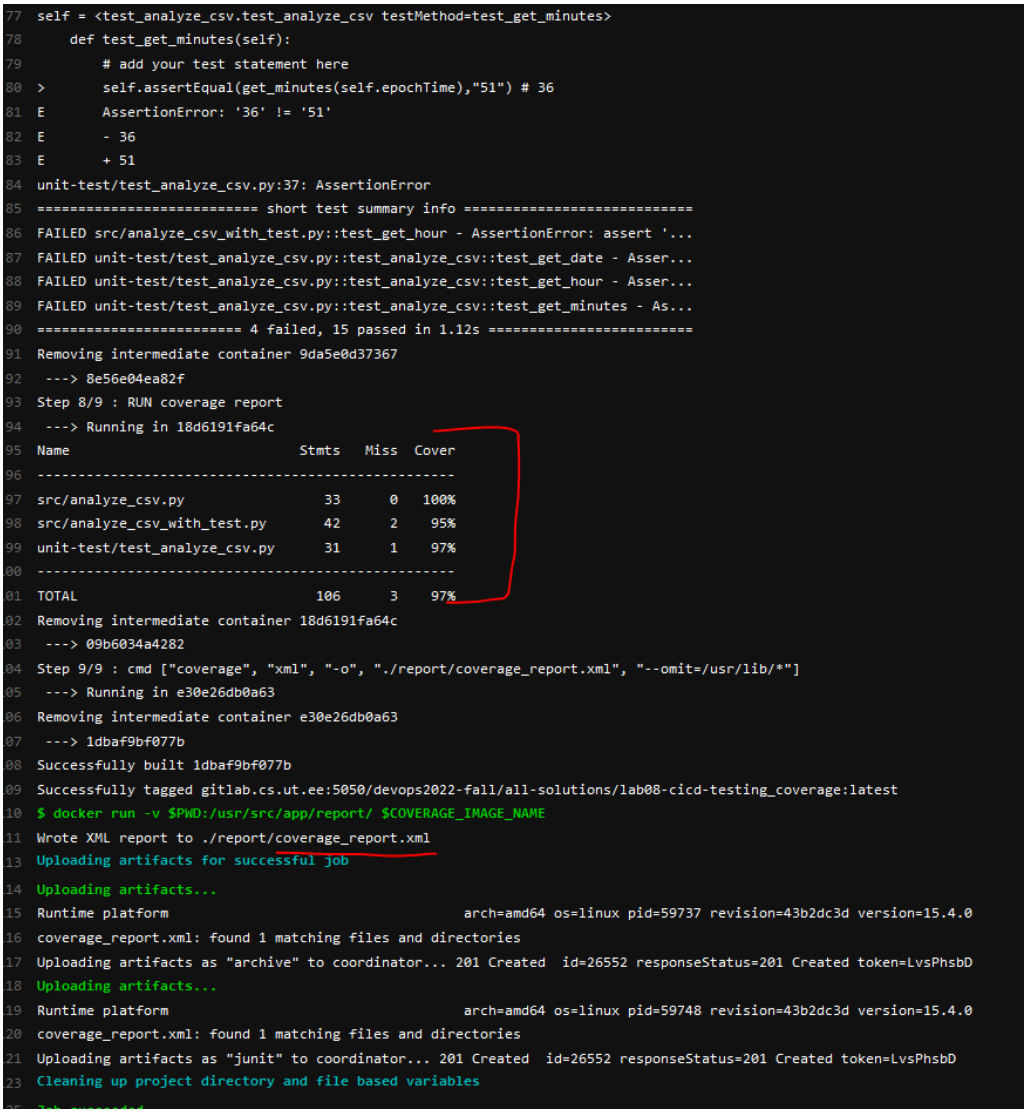
- *IMP:* You may need to update this file, if required.
- Now add the following ***job*** to the existing `.gitlab-ci.yml` configuration file.

```
# reference for python code coverage
#
https://docs.gitlab.com/ee/user/project/merge_requests/test_coverage_visualiza
tion.html#python-example
test_coverage_report_using_coverage:
  stage: test
  tags:
    - pytest
  script:
    - docker login -u dehury -p $gitlabpassword gitlab.cs.ut.ee:5050
    - docker build -t $COVERAGE_IMAGE_NAME -f ./docker/DockerfileTestCoverage
.
    - docker run -v $PWD:/usr/src/app/report/ $COVERAGE_IMAGE_NAME
  rules:
    - if: '$CI_PIPELINE_SOURCE == "push"'
    - if: '$CI_PIPELINE_SOURCE == "merge_request_event"'
      changes:
        - .gitlab-ci.yml
  artifacts:
    when: always
    expose_as: 'Test Coverage Report from other branch'
    paths: [coverage_report.xml]
    reports:
      junit: coverage_report.xml
```

- `$COVERAGE_IMAGE_NAME` Should be defined specifying the image name similar to the image name that you have defined in the previous practice session.
- Understand the `rules` keyword and update if required.
- Commit the above `DockerfileTestCoverage` and `.gitlab-ci.yml` changes to `cicd_test_coverage` branch. Don't create any merge request to the main branch.
    - Use the commit summary message "`STEP-2.3: Adding test coverage job to gitlab ci file in another branch.`"
- Now create a merge request to your own **`main`** branch using the gitlab web interface.
- Go to the list of `Merge requests`, and open the recent merge requests.
- Now you should see the following **similar** merge request status in the GitLab UI.



- Notice, there are two artifacts. The second one is "`Test Coverage Report`".
    - Click on the artifact name and you should be able to see the test coverage in `xml` format in a new browser tab.
    - Next to the artifact name, you can see the `Job name`. Click on Job name (in this case it should be `test_coverage_report_using_coverage`) to see the job logs. A sample job log is given below:



```
self = <test_analyze_csv.test_analyze_csv testMethod=test_get_minutes>
    def test_get_minutes(self):
        # add your test statement here
>       self.assertEqual(get_minutes(self.epochTime),"51") # 36
E       AssertionError: '36' != '51'
E       - 36
E       + 51
unit-test/test_analyze_csv.py:37: AssertionError
========================== short test summary info ==========================
FAILED src/analyze_csv_with_test.py::test_get_hour - AssertionError: assert '...
FAILED unit-test/test_analyze_csv.py::test_analyze_csv::test_get_date - Asser...
FAILED unit-test/test_analyze_csv.py::test_analyze_csv::test_get_hour - Asser...
FAILED unit-test/test_analyze_csv.py::test_analyze_csv::test_get_minutes - As...
======================== 4 failed, 15 passed in 1.12s ========================
Removing intermediate container 9da5e0d37367
 ---> 8e56e04ea82f
Step 8/9 : RUN coverage report
 ---> Running in 18d6191fa64c
Name                              Stmts   Miss  Cover
-----------------------------------------------------
src/analyze_csv.py                   33      0   100%
src/analyze_csv_with_test.py         42      2    95%
unit-test/test_analyze_csv.py        31      1    97%
-----------------------------------------------------
TOTAL                               106      3    97%
Removing intermediate container 18d6191fa64c
 ---> 09b6034a4282
Step 9/9 : cmd ["coverage", "xml", "-o", "./report/coverage_report.xml", "--omit=/usr/lib/*"]
 ---> Running in e30e26db0a63
Removing intermediate container e30e26db0a63
 ---> 1dbaf9bf077b
Successfully built 1dbaf9bf077b
Successfully tagged gitlab.cs.ut.ee:5050/devops2022-fall/all-solutions/lab08-cicd-testing_coverage:latest
$ docker run -v $PWD:/usr/src/app/report/ $COVERAGE_IMAGE_NAME
Wrote XML report to ./report/coverage_report.xml
Uploading artifacts for successful job
Uploading artifacts...
Runtime platform                                    arch=amd64 os=linux pid=59737 revision=43b2dc3d version=15.4.0
coverage_report.xml: found 1 matching files and directories
Uploading artifacts as "archive" to coordinator... 201 Created  id=26552 responseStatus=201 Created token=LvsPhsbD
Uploading artifacts...
Runtime platform                                    arch=amd64 os=linux pid=59748 revision=43b2dc3d version=15.4.0
coverage_report.xml: found 1 matching files and directories
Uploading artifacts as "junit" to coordinator... 201 Created  id=26552 responseStatus=201 Created token=LvsPhsbD
Cleaning up project directory and file based variables
```

Now merge the changes in `cicd_test_coverage` branch to the **`main`** branch.
It is recommended not to delete `cicd_test_coverage` branch after merging.

STEP-2.4: Find the code quality

Code quality gives a set of very important metrics such as the complexity, length of the code, etc. A low quality code also impacts the safety, security, and reliability of your codebase.

For this we will use the codeclimate tool. Using the Gitlab CI pipeline it is easy to find the quality of your code. Find the detailed steps here to use code climate in the GitLab CI pipeline.

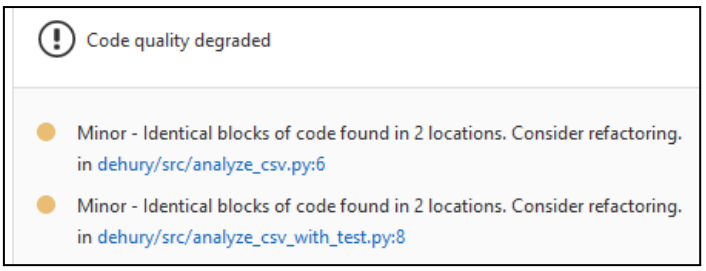
- Here, we will update the `.gitlab-ci.yml` configuration file in a new `cicd_test_quality` branch.
- GitLab CI provides a template `Code-Quality.gitlab-ci.yml`, that should be included as a template. For this, you need to add the following code snippet to the `.gitlab-ci.yml` configuration file.

```
include:
  - template: Code-Quality.gitlab-ci.yml
```

- `Code-Quality.gitlab-ci.yml` template file already defined a job named `code_quality`. We need to override some of the job descriptions with our own values. Add the following to the `.gitlab-ci.yml` configuration file.

```
code_quality:
    stage: test
    services:   # Shut off Docker-in-Docker
    tags:
        - codeclimate-shared-devops
    rules:
      - if: '$CI_PIPELINE_SOURCE == "push"'
      - if: '$CI_PIPELINE_SOURCE == "merge_request_event"'
        changes:
          - .gitlab-ci.yml
    artifacts:
        when: always
        expose_as: 'Code Quality Report from another branch'
        paths: [gl-code-quality-report.json]
```

- Commit the `.gitlab-ci.yml` configuration file changes in `cicd_test_quality` branch.
    - Use the commit summary as "`STEP-2.4: Added code-quality job.`"
- Now create again a merge request from `cicd_test_quality` branch to the **`main`** branch.
- When you create a merge request, the pipeline will start with the jobs in the `.gitlab-ci.yml` configuration file in `cicd_test_quality` branch.
- DIY: Fix, if there is any error in the ci file.
- Go to the `Merge requests` and open the recent merge request. Here you can see the merge request status.
    - Note that the test and the code quality report may take some time to appear.
    - If the `code_quality` job is running for the first time, this may take more time. Because internally two docker images of around 5GB each will be downloaded to the VM.
    - This job should be picked by the runner with tag `codeclimate-shared-devops`, if you have not created the second runner in the previous step by yourself. You can see this runner inside `Settings` -> `CICD` -> `Runners`.
    - [**Optional**] If you have created a second runner in your VM/Laptop/PC in the previous step, you may login to your system and issue `docker ps -a` command or `docker images` command.
- Explore the code quality degraded points and try to understand the reason behind this by yourself.



- At this point, the test report should have no further issue and you should see that green tick symbol.

Now merge the changes in `cicd_test_quality` branch to the **`main`** branch of your project. Don't delete the `cicd_test_quality` branch.

[Bonus task] Further exploring code quality metrics

This task is for you to explore the code quality metrics: such as excessive usage of nested if else statements, function with many arguments (more than 4), function with many return statements, etc.
The task is to

- Add a new `dummy.py` file inside the `src` folder. Create this in a new `cicd_test_bonus_task` branch.
- Add three functions:
  - A function with more than 6 arguments, e.g. `many_arg(a, b, c, d, e, f, g, h).`
  - A function with many nested if else statements.
  - A function with many return statements.
- Commit the code with the commit summary "`BONUS TASK: more dummy low quality functions.`"
- Create a merge request from `cicd_test_bonus_task` branch to the `main` branch.
- See the pipeline status and check the code quality report.
- Accept the merge request. That means now you have a `src/dummy.py` file in the **`main`** branch as well. Don't delete the `cicd_test_bonus_task` branch.

FINAL NOTE

Remember that, we may see the commit history while grading your submission. Further, please zip your GitLab project and submit through the course wiki page.

Deliverables

1- Download code of your GitLab project

2- Zip the code, screenshot and Upload the zip file to the course wiki page.

3- You may **Stop** the Virtual Machines and you can start using the same in the next **practice session.**

Don't delete your VMs

Reference:

1. https://docs.gitlab.com/ee/ci/unit_test_reports.html
2. List of CI templates: https://gitlab.com/gitlab-org/gitlab-foss/tree/master/lib/gitlab/ci/templates